



**US Army Corps  
of Engineers®**

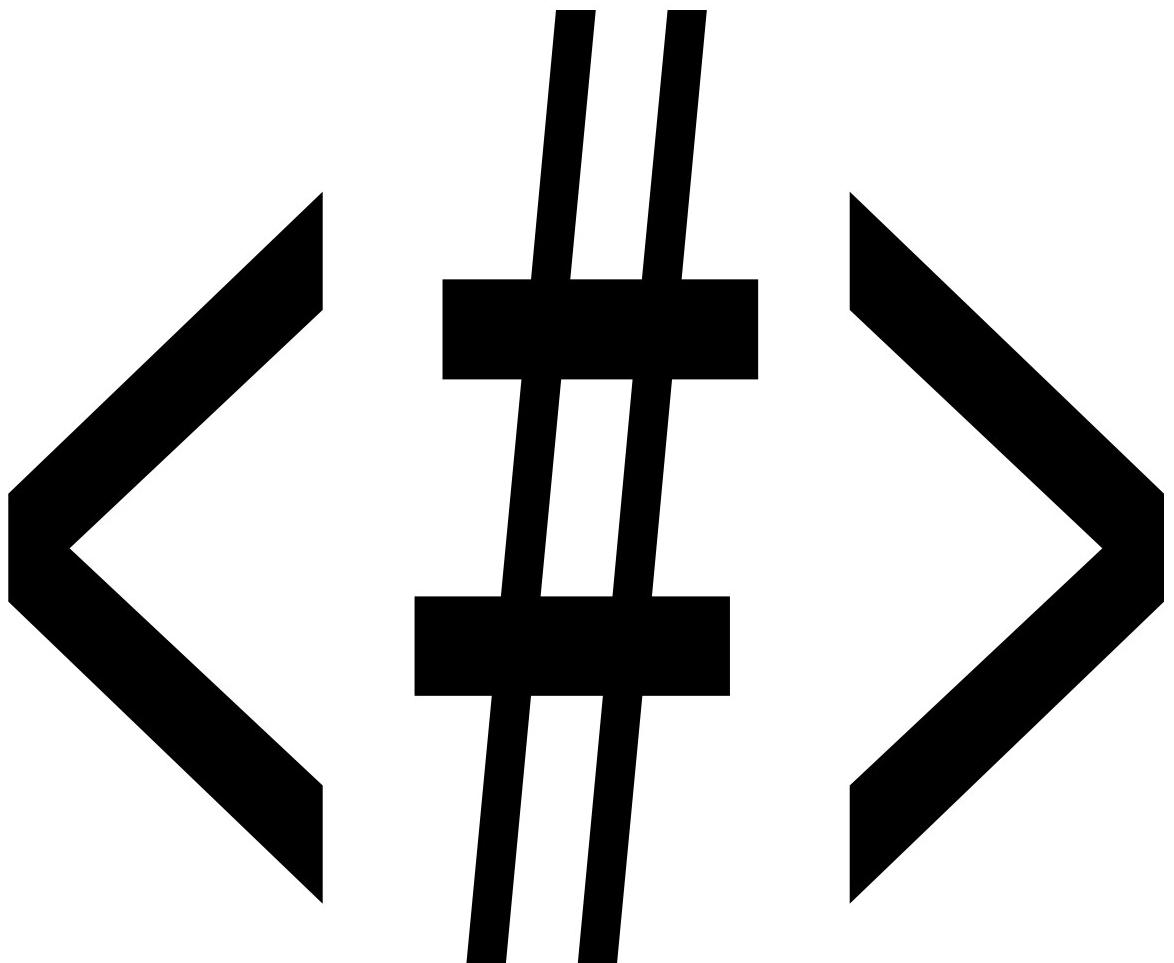
Engineering Research and  
Development Center

Flood & Coastal/NavSys/M.E. 6.2

## Flexible Scientific Software Distribution with Hashdist

D. S. Seljebotn, O. Čertík, A. R. Terrel, A. J. Ahmadi  
and C. E. Kees

November 20, 2013



<b>Report Documentation Page</b>			Form Approved OMB No. 0704-0188	
<p>Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p>				
1. REPORT DATE <b>20 NOV 2013</b>	2. REPORT TYPE	3. DATES COVERED <b>00-00-2013 to 00-00-2013</b>		
4. TITLE AND SUBTITLE <b>Flexible Scientific Software Distribution with Hashdist</b>			5a. CONTRACT NUMBER <b>W911NF-12-1-0604</b>	5b. GRANT NUMBER
			5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)			5d. PROJECT NUMBER	5e. TASK NUMBER
			5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) <b>Simula Innovation AS,P.O. Box 134,1325 Lysaker, Norway,</b>			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) <b>Engineer Research &amp; Development Center -International Research Office, ERDC-IRO, ATTN: Richmond, Unit 4507, APO, AE, 09421</b>			10. SPONSOR/MONITOR'S ACRONYM(S) <b>1535-EN-01</b>	
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT <b>Approved for public release; distribution unlimited</b>				
13. SUPPLEMENTARY NOTES				
14. ABSTRACT <p><b>Complex scientific software is often built on ?distributions?, or ?stacks? of software. Particulary as scientists and engineers work toward more integrated and therefore more interdependennt, scientific software, they need the ability to setup scientific stacks in many different user and hardware environments and hardware. Scientific stacks must be reproducible in the sense that ?regular? users should be able to install them and have them work as advertised. These stacks also need to work in challenging environments like cutting edge high performance commputers and handheld devices. Due to the authors? separate but similar experiences of the difficulty of buiding and maintaining scientific software stacks to support our own work, we worked together to build Hashdist, which is a a tool for building and managing custom software distributions based on a functional approach. It employs cryptographic hashing methods related to what are used in highly successful software source version control system to bring the same robustness to building and developing complex scientific software stacks.</b></p>				
15. SUBJECT TERMS				
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT <b>Same as Report (SAR)</b>	18. NUMBER OF PAGES <b>18</b>
a. REPORT <b>unclassified</b>	b. ABSTRACT <b>unclassified</b>	c. THIS PAGE <b>unclassified</b>	19a. NAME OF RESPONSIBLE PERSON	



# **Flexible Scientific Software Distribution with Hashdist**

**D. S. Seljebotn**

Simula Innovation AS

P.O. Box 134

1325 Lysaker, Norway

**O. Čertík**

Physics and Chemistry of Materials, T-1

Theoretical Division, MS B258

Los Alamos National Laboratory

Los Alamos, New Mexico 87545

**A. R. Terrel**

Computational Hydraulics Group

1 University Station, C0200

The University of Texas at Austin

Austin, TX 78712

**A. J. Ahmadia and C. E. Kees**

Coastal and Hydraulics Laboratory

U.S. Army Engineer Research and Development Center

3909 Halls Ferry Road

Vicksburg, MS 39180-6199

**Final Report**

Approved for public release; distribution is unlimited.

Prepared for U.S. Army Corps of Engineers  
Washington, DC 20314-1000

Under 7CJ429,LGBK72,622HK4,W911NF-12-1-0604

**Abstract:** Complex scientific software is often built on “distributions”, or “stacks”, of software. Particularly as scientists and engineers work toward more integrated, and therefore more interdependent, scientific software, they need the ability to setup scientific stacks in many different user and hardware environments and hardware. Scientific stacks must be reproducible in the sense that “regular” users should be able to install them and have them work as advertised. These stacks also need to work in challenging environments like cutting edge high performance computers and handheld devices. Due to the authors’ separate but similar experiences of the difficulty of building and maintaining scientific software stacks to support our own work, we worked together to build Hashdist, which is a tool for building and managing custom software distributions based on a functional approach. It employs cryptographic hashing methods related to what are used in highly successful software source version control system to bring the same robustness to building and developing complex scientific software stacks.

**Disclaimer:** The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. All product names and trademarks cited are the property of their respective owners. The findings of this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

**DESTROY THIS REPORT WHEN NO LONGER NEEDED. DO NOT RETURN IT TO THE ORIGINATOR.**

# Contents

<b>Preface</b> .....	<b>iv</b>
<b>1 Introduction</b> .....	<b>1</b>
<b>2 User's guide to Hashdist v. 0.2</b> .....	<b>2</b>
2.1 Installing and making the hit tool available.....	2
2.2 Setting up your software profile.....	2
2.3 Hashstack: collection of software profiles.....	3
2.4 Debug features.....	4
2.5 Developing the base profile.....	5
2.6 Further details.....	5
<b>3 Specifying a Hashdist software profile</b> .....	<b>6</b>
3.1 Profile specification .....	6
3.2 Package specifications.....	8
3.3 Conditionals.....	10
3.4 Stage system.....	11
3.5 Conclusions.....	12

## Preface

This report and the accompanying open source software are products of the Military Engineering 6.2 program, the Flood and Coastal Research and Development Program, the Navigation Systems Research Program, and the International Research Office, in collaboration with Simula Innovation, the University of Texas at Austin, and the Oak Ridge Institute for Science and Education. General supervision was provided by Mr. Jose E. Sanchez, Director, CHL; Dr. Charles A. Randall and Dr. Cary E. Talbot were project managers for this effort. Dr. David A. Horner was the Technical Director for Military Engineering and COL Jeffrey Eckstein was Commander and Executive Director of the Engineer Research and Development Center. Dr. Jeffrey P. Holland was Director.

# 1 Introduction

Hashdist is a platform independent package manager that builds from source. The core of the Hashdist library is the `hit` command line tool, which manages building of packages and installation of profiles. A profile is a collection of packages.

Each package has a cryptographic hash calculated using all inputs that are needed to build such a package (the package source code, the upstream package dependencies, and the configuration and build commands). As such, similar to how the version control system git hashes snapshots of the history of a source code project, a given package hash in hashdist uniquely identifies the package version in the history of building a software stack. Because the output of a config/build process depends on the source, their exact dependencies, their versions and many possible compiler or build options, hashdist hashes all of these inputs to identify the output of the build process uniquely.

Changing a build script of any package, source code tarball, or dependencies will result in a different hash, thus all packages that depend on it will get rebuilt. Once packages are built, they are stored in an artifact cache according to the hashdist package hash, so reverting the change results in instantenious build of the older version.

Besides the Hashdist program, we also provide a Hashstack repository with collection of software profiles that build on Linux, Mac, Windows and various clusters. Hashstack is distributed separately from Hashdist, so that users can easily provide or use their own collection of software profiles. That being said, Hashstack's goal is to provide most of the various Python Scientific packages that build using most of the various configurations that people use, so most users will probably just use Hashstack directly, or with minor modifications. Hashstack's goal is to be able to optionally reuse highly optimized libraries installed on a given cluster (BLAS, LAPACK, MPI, ...).

## 2 User's guide to Hashdist v. 0.2

### 2.1 Installing and making the *hit* tool available

Hashdist requires Python 2.7 and git.

To start using Hashdist, clone the repo that contains the core tool, and put the bin-directory in your PATH:

```
$ git clone https://github.com/hashdist/hashdist.git  
$ cd hashdist  
$ export PATH=$PWD/bin:$PATH
```

The hit tool should now be available. You should now run the following command to create the directory `~/.hashdist`:

```
$ hit init-home
```

By default all built software and downloaded sources will be stored beneath `~/.hashdist`. To change this, edit `~/.hashdist/config.yaml`.

### 2.2 Setting up your software profile

Using Hashdist is based on the following steps:

1. First, describe the software profile you want to build in a configuration file (“I want Python, NumPy, SciPy”).
2. Use a dedicated git repository to manage that configuration file
3. For every git commit, Hashdist will be able to build the specified profile, and *cache* the results, so that you can jump around in the history of your software profile.

Start with cloning a basic user profile template:

```
git clone https://github.com/hashdist/profile-template.git /path/to/myprofile
```

The contents of the repo is a single file `default.yaml` which a) selects a *base profile* to extend, and b) lists which packages to include. It is also possible to override build parameters from this file, or link to extra package descriptions within the repository (docs not written yet). The idea is to modify this repository to make changes to the software profile that only applies to you. You are encouraged to submit pull requests against the base profile for changes that may be useful to more users.

To build the stack, simply do:

```
cd /path/to/myprofile  
hit build
```

This will take a while, including downloading the source code needed. In the end, a symlink `default` is created which contains the exact software described by `default.yaml`.

Now, try to remove the `jinja2` package from `default.yaml` and do `hit build` again. This time, the build should only take a second, which is the time used to assemble a new profile.

Then, add the `jinja2` package again and run `hit build`. This exact software profile was already built, and so the operation is very fast.

When coupled with managing the profile specification with git, this becomes very powerful, as you can use git to navigate the history of or branches of your software profile repository, and then instantly switch to pre-built versions. [TODO: `hit commit`, `hit checkout` commands.]

Finally, if you want to have, e.g., release and debug profiles, you can create `release.yaml` and `debug.yaml`, and use the `-p` flag to `hit` to select another profile than `default.yaml` to build.

## 2.3 Hashstack: collection of software profiles

Hashstack is a collection of software profiles that builds on various architectures (Linux, Windows, Mac, clusters, ...) and allows optional reuse of system-wide packages (compilers, Lapack, Python, ...).

To build these profiles, you need the `hit` tool from Hashdist. Make sure that the `hit` command is in your path. To install the '`cloud.sagemath.yaml`' profile from Hashstack (that will work on Linux), do:

```
git clone https://github.com/hashdist/hashstack2
cd hashstack2
cp cloud.sagemath.yaml default.yaml
hit build
```

You can now for example run the IPython Notebook as follows:

```
default/bin/ipython notebook
```

On a Mac, you can create a profile by inheriting the `homebrew.yaml` profile.

## 2.4 Debug features

A couple of commands allow you to see what happens when building.

- Show the script used to build Jinja2:

```
hit show script jinja2
```

- Show the “build spec” (low-level magic):

```
hit show buildspec jinja2
```

- Get a copy of the build directory that would be used:

```
hit bdir jinja2 bld
```

This extracts Jinja2 sources to `bld`, puts a Bash build-script in `bld/_hashdist/build.sh`. However, if you go ahead and try to run it the environment will not be the same as when Hashdist builds, so this is a bit limited so far. [TODO: `hit debug` which also sets the right environment and sets the `$ARTIFACT` directory.]

## 2.5 Developing the base profile

If you want to develop the `hashstack2` repository yourself, using a dedicated local-machine profile repo becomes tedious. Instead, copy the `default.example.yaml` to `default.yaml`. Then simply run `hit build` directly in the base profile (in which case the personal profile is not needed at all).

`default.yaml` is present in `.gitignore` and changes should not be checked in; you freely change it to experiment with whatever package you are adding. Note the orthogonality between the two repositories: The base profile repo has commits like “Added build commands for NumPy 1.7.2 to share to the world”. The personal profile repo has commits like “Installed the NumPy package on my computer”.

## 2.6 Further details

*Specifying a Hashdist software profile*

## 3 Specifying a Hashdist software profile

There are specification file types in Hashdist. The *profile spec* describes *what* to build; what packages should be included in the profile and the options for each package. A *package spec* contains the *how* part: A (possibly parametrized) description for building a single package.

The basic language of the specification files is YAML, see <http://yaml.org>. Style guide: For YAML files within the Hashdist project, we use 2 space indents, and no indent before vertically-formatted lists (as seen below).

### 3.1 Profile specification

The profile spec is what the user points the *hit* tool to to build a profile. By following references in it, Hashdist should be able to find all the information needed (including the package specification files). An example end-user profile might look like this:

```
extends:
- name: hashstack
  urls: ['https://github.com/hashdist/hashstack2.git']
  key: 'git:5042aeaaee9841575e56ad9f673ef1585c2f5a46'
  file: debian.yaml

- file: common_settings.yaml

parameters:
  debug: false

packages:
  zlib:
  szlib:
  nose:
  python:
    host: true
  mpi:
    use: openmpi
  numpy:
```

```
skip: true

package_dirs:
- pkgs
- base

hook_import_dirs:
- base
```

### **extends:**

Profiles that this profile should extend from. Essentially this profile is merged on a parameter-by-parameter and package-by-package basis.

If anything conflicts there is an error. E.g., if two base profiles sets the same parameter, the parameter must be specified in the descendant profile, otherwise it is an error.

There are two ways of importing profiles:

- **Local:** Only provide the **file** key, which can be an absolute path, or relative to the directory of the profile spec file.
- **Remote:** If **urls** (currently this must be a list of length one) and **key** are given, the specified sources (usually a git commit) will be downloaded, and the given **file** is relative to the root of the repo. In this case, providing a **name** for the repository is mandatory; the name is used to refer to the repository in error messages etc., and must be unique for the repository across all imported profile files.

### **parameters:**

Global parameters set for all packages. Any parameters specified in the **packages** section will override these on a per-package basis.

Parameters are typed as is usual for YAML documents; variables will take the according Python types in expressions/hooks. E.g., `false` shows up as *False* in expressions, while '`false`' is a string (evaluating to *True* in a boolean context).

### **packages:**

The packages to build. Each package is given as a key in a dict, with a sub-dict containing package-specific parameters. This is potentially empty, which means “build this package with default parameters”. If a package is not present in this section (and is not a dependency of other packages) it will not be built. The **use** parameter makes use of a different package name for the package given, e.g., above, package specs for `openmpi` will be searched and built to satisfy the `mpi` package. The **skip** parameter says that a package should *not* be built (which is useful in the case that the package was included in an ancestor profile).

#### **package\_dirs:**

Directories to search for package specification files (and hooks, see section on Python hook files below). These acts in an “overlay” manner. In the example above, if one e.g., if searching for `python_package.yaml` then first the `pkgs` sub-directory relative to the profile file will be consulted, then `base`, and finally any directories listed in **package\_dirs** in the base profiles extended in **extends**.

This way, one profile can override/replace the package specifications of another profile by listing a directory here.

The common case is that base profiles set **package\_dirs**, but that overriding user profiles do not have it set.

#### **hook\_import\_dirs:**

Entries for `sys.path` in Python hook files. Relative to the location of the profile file.

## 3.2 Package specifications

Below we assume that the directory `pkgs` is a directory listed in **package\_dirs** in the profile spec. We can then use:

- Single-file spec: `pkgs/mypkg.yaml`
- Multi-file spec: `pkgs/mypkg/mypkg.yaml`, `pkgs/mypkg/somepatch.diff`,  
`pkgs/mypkg/mypkg-linux.yaml`

In the latter case, all files matching mypkg/mypkg.yaml and mypkg/mypkg-\*.yaml are loaded, and the **when** clause evaluated for each file. An error is given if more than one file matches the given parameters. One of the files may lack the **when** clause (conventionally, the one without a dash and a suffix), which corresponds to a default fallback file.

Also, Hashdist searches in the package directories for mypkg.py, which specifies a Python module with hook functions that can further influence the build. Documentation for the Python hook system is TBD, and the API tentative. Examples in base/autotools.py in the Hashstack repo.

Examples of package specs are in <https://github.com/hashdist/hashstack2>, and we will not repeat them here, but simply list documentation on each clause.

In strings; `{ {param_name} }` will usually expand to the parameter in question while assembling the specification needed, and are expanded before artifact hashes are computed. Expansions of the form  `${FOO}` are expanded at build-time (by the Hashdist build system or the shell, depending on context), and the variable name is what is hashed.

#### **when:**

Conditions for using this package spec, see rules above. It is a Python expression, evaluated in a context where all parameters are available as variables

#### **extends:**

A list of package names. The package specs for these *base packages* will be loaded and their contents included, as documented below.

#### **sources:**

Sources to download. For now, this should be a list with a single item, as implementing a **target** attribute is TBD.

#### **dependencies:**

Lists of names for packages needed during build (**build** sub-clause) or in the same profile (**run** sub-clause). Dependencies from base packages are automatically included in these lists, e.g., if `python_package` is listed in **extends**, then `python_package.yaml` may take care of requiring a build dependency on Python.

#### **build\_stages:**

Stages for the build. See Stage system section below for general comments. The build stages are ordered and then executed to produce a Bash script to run to do the build; the **handler** attribute (which defaults to the value of the **name** attribute) determines the format of the rest of the stage.

#### **when\_build\_dependency:**

Environment variable changes to be done when this package is a build dependency for *another* package. As a special case variable `ARTIFACT`

#### **profile\_links:**

A small DSL for setting up links when building the profile. What links should be created when assembling a profile. (In general this is dark magic and subject to change until documented further, but usually only required in base packages.)

### 3.3 Conditionals

The top-level **when** in each package spec has already been mentioned. In addition, there are two forms of local conditionals within a file. The first one can be used within a list-of-dicts, e.g., in **build\_stages** and similar sections:

- `when: platform == 'linux'`  
`name: configure`  
`extra: [--with-foo]`
  
- `when: platform == 'windows'`  
`name: configure`  
`extra: [--with-bar]`

The second form takes the form of a more traditional if-test:

```
- name: configure
  when platform == 'linux':
    extra: [--with-foo]
  when platform == 'windows':
    extra: [--with-bar]
  when platform not in ('linux', 'windows'):
    extra: [--with-baz]
```

The syntax for conditional list-items is a bit awkward, but available if necessary:

```
dependencies:
build:
- numpy
- when platform == 'linux': # ! note the dash in front
  - openblas
- python
```

This will turn into either [numpy, python] or [numpy, openblas, python]. The “extra” – is needed to maintain positioning within the YAML file.

## 3.4 Stage system

The **build\_stages**, **when\_build\_dependency** and **profile\_links** clauses all follow the same format: A list of “stages” that are partially ordered (using **name**, **before**, and **after** attributes). Thus one can inherit a set of stages from the base packages, and only override the stages one needs.

There’s a special **mode** attribute which determines how the override happens.  
E.g.:

```
- name: configure
  mode: override # !!
  foo: bar
```

will pass an extra `foo: bar` attribute to the configure handler, in addition to the attributes that were already there in the base package. This is the default behaviour. On the other hand,:

```
- name: configure
  mode: replace  # !!
  handler: bash
  bash: |
    ./configure --prefix=$ARTIFACT
```

entirely replaces the configure stage of the base package. Finally,:

```
- name: configure
  mode: remove  # !!
```

removes the stage.

### 3.5 Conclusions

HashDist and HashStack are unique packages in the scientific community today. They address persistant barriers to the progress of computational science and engineering, particularly barriers to reproducibility in large-scale calculations and barriers to sharing and integrating development efforts across groups and institutions.